Model-View Sensor Data Management in the Cloud

Tian Guo, Thanasis G. Papaioannou and Karl Aberer
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland 1015
Email: firstname.lastname@epfl.ch

***Abstract*—Infinite nature of sensor data poses a serious challenge for query processing even in a cloud infrastructure. Model-based sensor data approximation reduces the amount of data for query processing, but all modeled segments need to be scanned, in the worst case. In this paper, we propose an innovative index for modeled segments in key-value stores, namely KVI-index. KVI-index has an in-memory tree component and a secondary structure materialized in the key-value store that maps the tree nodes to the modeled data segments. Then, we introduce a KVI-index-Scan-MapReduce hybrid approach to perform efficient query processing. As proved by a series of experiments in a real private cloud infrastructure, our approach outperforms in query response time and index updating efficiency both Hadoop-based parallel processing of the raw sensor data and multiple alternative indexing approaches of model-view data.**

***Index Terms*—index, key-value, MapReduce, approximation, query processing**

I. Introduction

Recent advances in sensor technology have enabled the vast deployment of sensors embedded in user devices that monitor various phenomena for different applications of interest, e.g., air/electrosmog pollution, radiation, early earthquake detection, soil moisture, permafrost melting, etc. Sensors continuously produce unbounded time series of measurements that can be erroneous or have missing values posing great challenges for data management. To this end, various model-based sensor data management techniques [1]–[4] have been proposed. Models exploit the inherent correlations (e.g. with time or among data streams) in time series to split data in segments and approximate each segment with a certain mathematical function derived by the model within a certain error bound. These techniques aim to facilitate query processing by accessing or generating minimal amount of data [2], [5] and deal with missing values in data by an abstraction layer over the sensor data [3].

However, proposed model-based query processing approaches [2], [3] mostly employ the relational data model and process queries based on materialized views or interval indices [6] on top of modeled segments of sensor data. Alternatively, in the cloud era, time series are stored in key value stores [7] and query processing is parallelized via MapReduce.

In this paper, we exploit key-value stores and the MapReduce parallel computing paradigm, two significant aspects of cloud computing, to realize indexing and querying model-view sensor data in the cloud. One modeled segment is characterized by its time and value intervals [2]–[4]. In order to process range or point queries on model-view sensor data [8], our index in the cloud store should excel in processing interval data. Current key-value built-in indices do not support interval related operations. The interval index for sensor data management should not only work on static data set, but it should be dynamically updated based on the new arriving segments of sensor data [9]. If traditional batch-updating or periodical re-building strategy is applied here [10], then the high speed of sensor data generation may lead to a large size of the new unindexed data set, even in short time, with significant index updating delay. The performance of queries involving both indexed and unindexed data will degenerate greatly, thus the interval index in the cloud store should be able to be updated based on individual modeled segments in an online manner.

The contributions of this paper can be summarized as follows:

- Innovative interval index: We propose an innovative interval index for model-view based sensor data management in key-value stores, referred to as *KVI-index*. *KVI-index* is a two-tier structure consisting of one lightweight and memory-resident binary search tree and one index-model table materialized in the key-value store. This composite index structure can dynamically accommodate new sensor data segments very efficiently.
- Hybrid modeled-segment query processing: After exploring the search operations in the in-memory structure of the KVI-index for range and point queries that locate modeled segments that may satisfy the query, we introduce a hybrid query processing approach that integrates range scan and MapReduce to process these segments in parallel and identify the qualified ones.
- Intersection search: We introduce an enhanced intersection search algorithm (*iSearch+*) that produces consecutive results suitable for MapReduce processing. We theoretically analyze the efficiency of (*iSearch+*) and find the bound on the redundant index nodes that it returns.
- Experimental evaluation: Our framework has been fully implemented, including on-line sensor data segmentation, modeling, KVI-index and the hybrid query processing, and it has been thoroughly evaluated against a significant number of alternative approaches. As experimentally shown based on real sensor data, our approach significantly outperforms in terms of query response time and index updating efficiency all other ones for answering time/value point and range queries.

The remainder of this paper is as follows: Sec. II summarizes some related work on model-view sensor data management, interval index and index-based MapReduce optimization approaches. In Sec. III, we provide a brief description about sensor data segmentation, querying model-view sensor data and the necessity to develop interval index for managing model-view sensor data in key-value stores. The detailed designs of our innovative KVI-index and the hybrid query processing approach are discussed in Sec. IV and V respectively. Then, in Sec. VI, we present thorough experimental results to evaluate our approach with traditional query processing ones on both raw sensor data and modeled data segments. Finally, in Sec. VII, we conclude our work.

II. Related Work

Many researchers have proposed techniques for managing modeled segments of sensor data in relational databases. MauveDB [3] designed a model-based view to abstract underlying raw sensor data; it then used models to project the raw sensor readings onto grids for query processing. As opposed to MauveDB, FunctionDB [2] only materializes segment models of raw sensor data. Symbolic operators are designed to process queries using models rather than raw sensor data. However, both approaches in [3] and [2] do not take into account the role of an index in managing modeled segments of sensor data.

Many relevant index structures [6], [9], [10] have been proposed to manage interval data [11]. However, they all have memory-oriented structure and cannot be directly applied to the distributed environment of key-value cloud stores. Many efforts [8]–[11] have also been done to externalize these in-memory index data structures. The relational interval tree(RI-tree) [6] integrates interval tree into relational tables and transforms interval queries into SQL queries on tables. This method makes efficient use of built-in B+-tree indices of RDBMS. In [12], they proposed an approach that enables key-value stores to support query processing of multi-dimensional data via integrating space filing order into row-keys. The latest effort for supporting interval index in key-value stores [10] utilizes MapReduce to construct a segment tree materialized in the key-value store. This approach outperforms the interval query processing provided by HBase (http://hbase.apache.org/) and Hive (http://hive.apache.org/). However, segment tree [10] is essentially a static interval index, as is also the case with [13]. Therefore, a segment tree re-building phase needs to be periodically executed to include new data.

MapReduce parallel computing is an effective tool to access large scale of segment models of sensor data in cloud stores. Many researchers proposed index techniques to avoid data scan by MapReduce for low-selective queries [14]. The authors in [15] integrate indices into each file split, so that mappers can use index to only access predicate qualified data records. In [16], indices applicable to queries with predicates on multiple attributes via indexing different record attributes in different block replicas were designed. In [17], a split-level index is designed to decrease MapReduce framework overhead. As compared to record-level optimizations in [15], [16], split-level indices eliminate irrelevant file splits before launching mappers, and thus the data transferring and starting up overheads are further saved.

III. Overview of model-view sensor data management

In this section, we discuss the issues for managing modeled segments of sensor data in a key-value store. First, we explain what is modeled data segments. Then, we discuss possible storage schemas for modeled segments and explain the necessity to develop a segment model index therein. Last, we describe the query types of our focus and some particular techniques for processing model-view sensor data queries.

A. Sensor data segmentation

Among the objectives of sensor data modeling are to compress the raw data, to fill missing values and to detect data outliers. A typical modeling approach fragments the data stream into *modeled data segments*, and then approximates each data segment by a mathematical function with certain parameters [18], so that a specific error norm is satisfied. Query processing can then be performed on the materialized modeled data segments instead of the raw sensor data, as in [2]. Sensor data segmentation and modelling have been extensively studied in [4], [18], [19].

B. Storage model

One idea for storing modeled segments in key-value stores could be to do it similarly to that of the raw sensor data storage such as Open-TSDB [7], and takes the time interval of one segment as the row key (rk). As rows are sorted on the row key in the key-value store, the starting points of time intervals are in ascending order. Therefore, although for time range or point queries, the query processor knows when to stop the scan, it still needs to start the scan from the beginning of the table. The same problem happens to the table with value intervals as row-keys. In summary, simply incorporating time, value interval or model coefficients into the row-key cannot help accelerate the query processing. A generic key-value based interval index for sensor data segments is necessary. Moreover, key-value represented interval index can take advantage of efficient key based random access, range scan and parallel computing of key-value stores [20].

C. Querying model-view sensor data

In this paper, we focus on the following four types of fundamental queries on model-view sensor data.

- time point query: return the value of one sensor at a specific time point.
- value point query: return the timestamps when the value of one sensor is equal to the query value. There may be multiple time stamps of which sensor values satisfy the query value.
- time range query: return the values of one sensor during the query time range.
- value range query: return the time intervals of which the sensor values are within the query value range. There may be multiple time intervals of which sensor values satisfy the query value range.

The generic process to query model-view sensor data queries comprises the following two steps:

- Searching of qualified modeled segments: The qualified modeled segments are defined as the ones of which time (resp. value) intervals intersect the query time (resp. value) range or point. This step should make use of an interval index to localize all the qualified modeled segments in the segment model store.
- Model gridding: In model-view sensor data management system, only modeled segments are usually materialized, instead of raw data values. Qualified modeled segments are too abstract and a finite set of data points are more useful as query results [2]. Therefore, model gridding is another necessary process [2], [3]. Model gridding step applies three operations to each qualified segment: (i) It discretizes the time interval of the segment at a specific granularity to generate a set of time points. (ii) It generates the sensor values at the discrete time points based on the model that approximates the segment. (iii) It filters out the sensor data that does not satisfy the query predicates. The qualified time or value points from gridding all qualified modeled segments are returned as query results.

IV. Key-Value Interval Index

We will first present the design of the two-tier model index on key-value stores then we will discuss the updating algorithm of the model index.

A. Structure of KVI-index

We propose the **k**ey-**v**alue represented **i**nterval index (KVI-index) to index time and value intervals of modeled data segments. The interval tree is chosen here, because in-memory interval tree's primary-secondary structure is convenient for externalization to the key-value store [6]. Furthermore, the searching and scanning algorithm of the interval tree to process queries is fit for being implemented via the MapReduce computing paradigm to enhance query processing performance.

Our KVI-index is a novel in-memory and key-value composite index structure. The virtual searching tree(*vs-tree*) resides in memory, while an index-model table in the key-value store is devised to materialize the secondary structure(*SS*) of each node in *vs-tree*.

1) In-memory structure: The in-memory virtual searching tree(*vs-tree*) is a standard binary search tree shown in Fig. 1(a). Each time (or value) interval is registered on only one node of *vs-tree*, which is the one with the interval first overlaps along the searching path from root. This node is defined as registration node τ for this interval. Each node of *vs-tree* has an associated secondary structure(*SS*), materialized in the key-value store, which stores the substantial information of the modeled segments registered at this node.

We apply space-partition strategy for *vs-tree*. The height of the *vs-tree* is denoted by h. We set the value of leftmost leaf node as 0. For negative sensor data values, we use simple shifting to have the data range start from 0 for convenience. Then, the domain of the *vs-tree* is $[0, R]$ and $R = 2^{(h+1)} - 2$. The value of root node is $r = 2^h - 1$. During the whole life of KVI-index, only the root value r is kept, since, due to the lightweight computability of the space-partition, the value of each node in the searching path from the root to the node that has the queried point or interval can be calculated in run time. All the operations on *vs-tree* are performed in memory and are thus very efficient. As the domains of time and value of the sensor data are different, two *vs-tree*s one for times and another for values are kept in memory simultaneously for answering time and value queries respectively.

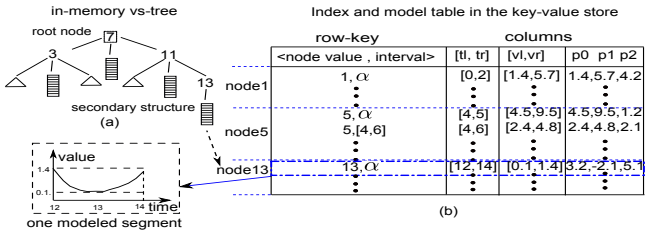


Fig. 1. Two-tier structure of KVI-index

2) Index-model table: We designed a novel index-model composite storage schema, which enables one table not only to store the modeled segments, but also to materialize the structural information of the *vs-tree*, i.e., the *SS*s for each tree node.

The index-model table is shown in Fig.1(b). Each row corresponds to only one modeled segment of sensor data, e.g., the data segment shown in the black dotted rectangle in Fig.1(c). A row key consists of the node value and the interval of an indexed modeled segment at that node. One modeled segment's time, value interval and coefficients are all stored in different columns of the same row. So far, the *SS*s of each node correspond to a consecutive range of tuples in the index-model table. For instance, the rows corresponding to the *SS*s of node 1, node 5 and node 13 in *vs-tree* are illustrated in Fig. 1(b). Analogously, we have two index-model tables that correspond to time and value *vs-tree*s respectively.

B. KVI-index updates

The complete modeled segment updating algorithm of KVI-index is shown in the Alg. 1. It includes two processes:

(1) *Registration node searching*: localize the node τ at which one time(value) interval $[l_t, r_t]$ should be registered.

(2) *Materialization of modeled segments*: construct the row-key based on the τ and materialize one modeled segment's information into the columns of corresponding row.

1) Registration node searching (rSearch): This algorithm first involves a *domain expansion process* to dynamically adjust the domain of the *vs-tree* according to the domain variation of the sensor data. Then, the registration node can be found on the validated *vs-tree*.

Lemma 1: For a modeled segment M_i with time(value) interval $[l_t, r_t]$, its registration node lies in a tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$.

Proof: The domain of one tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$ is $[0, 2^{\lceil \log(r_t+2) \rceil} - 2]$. As $2^{\lceil \log(r_t+2) \rceil} - 2 \geq 2^{\log(r_t+2)} - 2 = r_t$, the registration node of interval $[l_t, r_t]$ must be in a tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$. ■

Lemma 2: For a modeled segment M_i with time (value) interval (l_t, r_t), if the right end-point r_t satisfies $r_t > R$, the domain of *vs-tree* needs to expand.

Algorithm 1: Time (or value) KVI-index Updating

Input: $[l_v, r_v]$, $[l_t, r_t]$, /* value and time intervals of one modeled segment M_i, r /* M_i denotes the coefficients of the modeled segment

```
begin
    /* dynamic domain expansion
    if (r_t > R) then
        r= 2^⌈log(r_t+2)⌉−1 − 1 /* expand to new root value
    /* registration node search
    node=r; h=log(r)-1;
    while (h ≥ 0) do
        if (l_t ≤ node and r_t ≥ node) then
            break; /* node is the registration node
        else
            if (l_t > node) then
                node= node+ 2^h;
            if (r_t < node) then
                node= node- 2^h;
            h=h-1;
    /* materialized into the index-model table.
    if the SS of node has been initialized then
        rowkey= <node|l_t|r_t>;
    else
        rowkey= <node|α>;
    insert [l_v, r_v], [l_t, r_t], M_i into the table.
```

Proof: The current *vs-tree*'s height is h and $l_t \leq r$, the interval $[l_t, r_t]$ rides over the root node r. Assume we don't expand the domain and hang $[l_t, r_t]$ on r. When a new model M_j with a time (or value) interval $[l_t^{'}, r_t^{'}]$ and $l_t^{'} > R$ comes, the root value has to increase to $r^{'}$=$2^{\lceil \log r_t^{'}+2 \rceil -1}$-1 as $[l_t^{'}, r_t^{'}]$ intersects no node of current *vs-tree*. Then between r and $r^{'}$, there is one node with value $2^{(h+1)}$-1. The interval (l_t, r_t) is stored at node $r = 2^h$-1 and $r_t > 2^{(h+1)}$-2 $\Rightarrow r_t \geq 2^{(h+1)}$-1, therefore registering the interval $[l_t, r_t]$'s registration on node r contradicts with the interval registration rule. ■

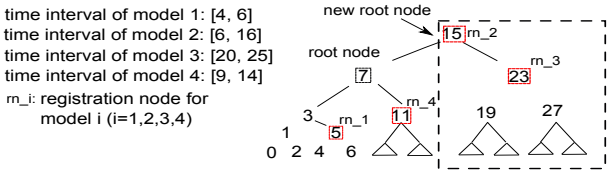


Fig. 2. Registration node searching of KVI-index

Using Lemma1 and Lemma2, KVI-index is able to dynamically decide when and how to adjust the domain $[0, R]$ of *vs-tree*. The complete *rSearch* can be illustrated by Fig. 2. When $model1$ is to be inserted, the *vs-tree* rooted at $node7$ is still valid. $model1$ is registered at $node5$. However, when $model2$ arrives, its right end-point, i.e., 16, exceeds the domain $[0, 14]$. The *vs-tree* is expanded having 15 as new root and the new extended domain is the area enclosed by the dotted block in Fig. 2. Then, the sub-sequential $model2$ and $model3$ can be updated successfully.

2) Materialization of modeled segment: When materializing model M_i into the *SS* of a node τ, the row-key may be chosen in two ways:

- Upon initialization of the *SS* of node τ: when no modeled segment has been stored at τ's *SS*, the row key is chosen as $< \tau, \alpha >$ for model M_i. Here, α is a postfix of row key to indicate that this row is the starting position of τ's *SS* in the table.
- Upon updating the *SS* of node τ: when the *SS* of τ has already been initialized, the time interval $[l_t, r_t]$ (resp. $[l_v, r_v]$ for value interval) to be indexed will be incorporated into the row key, i.e., $< \tau, l_t, r_t >$ (resp. $< \tau, l_v, r_v >$ in the index-model table for values). In this way, different modeled segments stored in the same *SS* of a node do not overwrite each other.

The selection of specific α should make sure that the binary representation of $< \tau, \alpha >$ is in front of any other $< \tau, l_t, r_t >$. This design is useful for query processing. For instance, if the query processor requires to access all the modeled segments stored at registration node 5, then we know that all the corresponding modeled segments lie in the rows within the row-key range $[< 5, \alpha >, < 6, \alpha >)$. For example, take the $model1$ and $model2$ in Fig. 3. First, the KVI-index checks whether the starting modeled segments of $node5$ and $node15$, namely rows with key $< 5, \alpha >$ and $< 15, \alpha >$, exist. Then, the row key $< 5, 4, 6 >$ is constructed for $model1$ as the *SS* of $node5$ has been initialized, whilst KVI-index constructs the row key $< 15, \alpha >$ for $model2$.

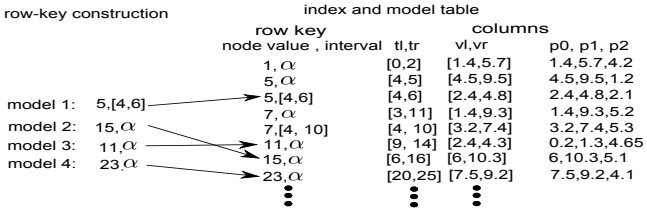


Fig. 3. Modeled segment materialization of KVI-index

V. Query Processing via KVI-index and MapReduce

For querying model-view sensor data, the searching process of qualified modeled segments (defined in Sec.III) in KVI-index includes two steps:

- Intersection and point search: The intersection search on *vs-tree* is used for range queries, while point search is employed for point queries. They are responsible for collecting the nodes that accommodate qualified modeled segments [6], [9] in their secondary structures *SS*s.
- Modeled-segment filtering: Due to the rule for interval registration at the nodes of the *vs-tree*, the *SS* of a node may contain some intervals irrelevant to queried range or point [6], [8], [13]. In KVI-index, the *SS*s of all nodes found by the search operation are accessed to filter out unqualified segments.

After the above two steps, model gridding component fetches the coefficients of each qualified modeled segment and performs gridding. Next, we first describe an enhanced intersection search algorithm on *vs-tree* that benefits KVI-Scan-MapReduce query processing introduced later in this section. We then present the point search algorithm on *vs-tree*. Subsequently, we introduce our novel hybrid KVI-Scan-MapReduce query processing. Last, we theoretically analyze the enhanced intersection search algorithm of the KVI-index.

A. Intersection and point search

1) Enhanced interval intersection search: Alg. 2 presents the $iSearch^+$. Given a time (resp. value) range query $[l_t, r_t]$, $iSearch^+$ first calls the *rSearch* to find the registration node τ of $[l_t, r_t]$. The nodes on the searching path from the root node to the one preceding τ form a node set denoted by $\mathcal{S}_0$. The $iSearch^+$ stops at the node, which is closest to the left-end point l_t. All the nodes along the left-descending path form a node set, denoted by $\mathcal{S}_l$, while the node with the minimum value in this path is denoted by l_s. Analogously, $\mathcal{S}_r$ is the node set from the right-descending path and r_s is the node with the maximum value in this path. Any node outside the range $[l_s, r_s]$ and the set $\mathcal{S}_0$ does not have any qualified modeled segments.

Algorithm 2: $iSearch^+$ of *vs-tree*

```
   Input: time query range [l_t, r_t], root value r
   Output: node set S_0 and D
1  begin
2      /* construct S_0
3      node=r; h=log(r)-1;
4      while (h ≥ 0) do
5          if (l_t ≤ node and r_t ≥ node) then
6              break; /* node is the registration node
7          else
8              S_0 = S_0 ∪ node
9              if (l_t>node) then
10                 node= node + 2^h;
11             if (r_t<node) then
12                 node= node - 2^h;
13             h=h-1;
14     /* construct D .
15     l_s= 2^⌊log(l_t)⌋, r_s= R - 2^⌊log(R−r_t)⌋, D = [l_s, r_s]
```

The traditional intersection search would return the node set $\mathcal{C} = \mathcal{S}_0 \cup \mathcal{S}_l \cup \mathcal{S}_r \cup [l_t, r_t]$ for further modeled-segment filtering and gridding. Our $iSearch^+$ outputs the discrete node set $\mathcal{S}_0$ and a consecutive range of nodes $\mathcal{D} = [l_s, r_s]$. For example, take the range query in Fig. 4(a). $node7$ is the registration node of query range $[6, 10]$. The traditional $iSearch$ returns the discrete node sets shown in the solid boxes of Fig. 4(a), while our $iSearch^+$ returns a range of nodes $[3, 11]$ and $\mathcal{S}_0 = \{15\}$. We will see how the output of $iSearch^+$ benefits the hybrid query processing later in Subsection V-B.

2) Point search: We denote the point search by *sSearch* as it functions as the stabbing search of interval tree. The *sSearch* is a binary search that records the nodes along the descending path. We present the *sSearch* in Fig. 4(a). For example, when querying the sensor value of time point 24, the node set $\mathcal{S}_0 = \{15, 7, 11, 9, 10\}$ is returned by *sSearch*. Since there is no split searching, as in $iSearch^+$, only one node set is produced here. We denote this node set by $\mathcal{S}_0$ as well, so as to facilitate the description of the hybrid KVI-Scan-MapReduce query processing that follows next.

B. KVI-Scan-MapReduce query processing

We first analyze the location distribution of the *SS*s of the nodes found by $iSearch^+$ and *sSearch* in the index-model table. The characteristics of this distribution inspired us to propose the hybrid KVI-Scan-MapReduce query processing approach.

1) SS location distribution: There are two cases for the *SS* distribution in the index-model table, described below.

- $\mathcal{S}_0$: The *SS*s of $\mathcal{S}_0$ are non-consecutive and sparsely distributed in the index-model table. The node value is the primary part of the row-key; thus, the distance between *SS*s of $\mathcal{S}_0$ depends on the numerical difference of node values. As $\mathcal{S}_0$ includes the nodes from root node to the one preceeding the τ, the intra-distances between any consecutive nodes in $\mathcal{S}_0$ are 2^{h-i}, where $i = 0, \cdots, h - d_\tau$ is the position of the node in the descending search path $\mathcal{S}_0$ and d_τ is the depth of τ. Obviously, the intra-distances in $\mathcal{S}_0$ are greater than those for other nodes below τ in the search path.
- $\mathcal{D}$: The *SS*s of $[l_s, r_s]$ are clustered around the ones of $[l_t, r_t]$ in the index-model table. The *SS*s of $[l_t, r_t]$ are all adjacent in the index-model table. The *SS*s of $[l_s, r_s]$ are bounded by those of the sub-tree rooted at τ and the nodes in $[l_s, r_s]$ are a superset of the nodes in $[l_t, r_t]$. The deeper the registration node τ is located, the tighter the set of the *SS*s of $[l_s, r_s]$ over those of $[l_t, r_t]$.

For example, take the time (or value) query range $[6, 10]$ in Fig. 4(a). The registration node is $node7$. Then, $\mathcal{S}_0 = \{15\}$ and $\mathcal{D} = [3, 11]$. The sub-tree rooted at $node7$ covers the node range $\mathcal{E} = [0, 14]$ and $\mathcal{D} \subset \mathcal{E}$. From Fig. 4(b), the *SS*s of $\mathcal{D}$ are clustered around those of $[6, 10]$ and bounded by the *SS*s of $\mathcal{E}$. However, $node15$'s *SS* is located far away from those of $[3, 11]$.

If *SS*s of $\mathcal{S}_0$ and $\mathcal{D}$ are processed via straightforward random access and range scan provided by key-value stores, the entire modeled-segment filtering and gridding processes are conducted locally at the application side. For a table of multiple or hundreds of GBs, the communication and computation costs are prohibitively high for the application side even for low-selective queries.

The modeled segment filtering-gridding processing matches MapReduce's filtering-aggregation paradigm. Considering the research results from [15]–[17], for CPU non-intensive workload, I/O cost, network latency and starting-up overhead of mappers are dominant in the execution time of MapReduce programs. If the *SS*s of $\mathcal{S}_0$ and $\mathcal{D}$ are all processed by MapReduce, a lot of time is wasted for mappers that process irrelevant *SS*s in the index-model table. This is because MapReduce will access the continuous regions of the table including the *SS*s of nodes between the $\mathcal{S}_0$ and $\mathcal{D}$ due to the sequential data feeding mechanism in the mapping phase. For example, in Fig. 4(b), the *SS*s of $\mathcal{D}$=$[3, 11]$ and $\mathcal{S}_0$=$\{15\}$ are distant in the table. Hence, MapReduce will launch many un-necessary mappers for the irrelevant *SS*s of nodes between 11 and 15, in order to process the *SS*s of $\mathcal{S}_0$ and $\mathcal{D}$.

2) Hybrid model filtering and gridding: As discussed above, simply using range scan or MapReduce to process *SS*s are both problematic. Our idea is to design a hybrid KVI-Scan-MapReduce paradigm that combines range scan and MapReduce for processing *SS*s, as follows:

- (1) $\mathcal{S}_0$: the height of *vs-tree* is bounded by $log(R)$, and thus the amount of computation on $\mathcal{S}_0$ is limited. As the

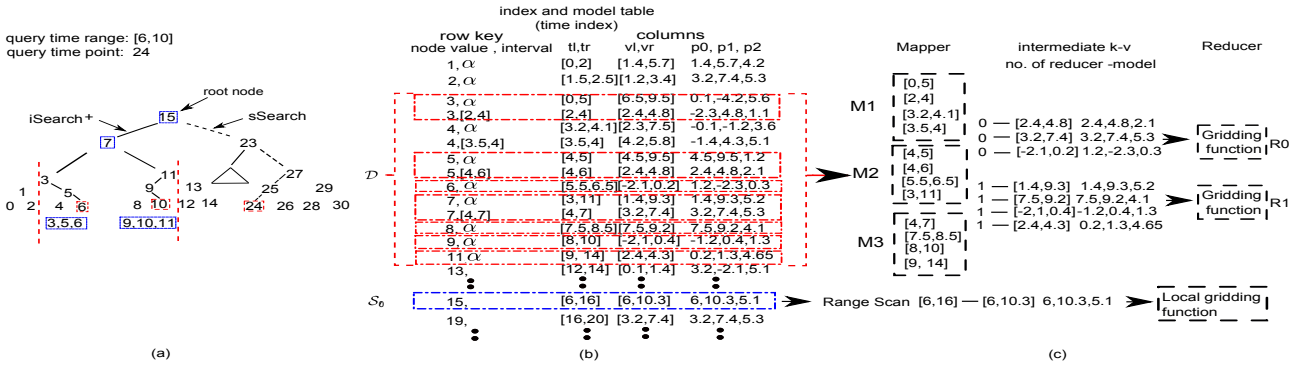


Fig. 4. Workflow of KVI-Scan-MapReduce approach. (a) $iSearch^+$ and *sSearch*.(b) *SS* location distribution for the range query. (c) hybrid processing

*SS*s of $\mathcal{S}_0$ are sparsely distributed in the index-model table and each *SS* of $\mathcal{S}_0$ can be considered as a small range of clustered index, the random-access- and range-scan-based model filtering and gridding is suitable.

- (2) $D = [l_s, r_s]$: the successive range $[l_s, r_s]$ delimits a tight boundary of the sub-index-model table over the relevant *SS*s that are suitable for processing with MapReduce.

This hybrid paradigm eliminates the Map-phase processing of *SS*s of irrelevant nodes between $\mathcal{S}_0$ and $\mathcal{D}$ and the nodes between the elements of $\mathcal{S}_0$. Moreover, it is non-intrusive for both the key-value store and MapReduce. Regarding the time (or value) point query, it only produces the node set $\mathcal{S}_0$ without $\mathcal{D}$, hence, only range-scan-based model filtering and gridding is needed.

Suppose that the number of reducers is P and each reducer is denoted by $0, \cdots, P$-1. For range queries, the partition function f is used to assign the qualified modeled segments into different reducers. It is designed on the basis of query time (resp. value) range $[l_t, r_t]$ (resp. $[l_v, r_v]$) and the time (or value) interval $[l_i, r_i]$ of each modeled segment i. The idea is that each of the reducers is in charge of one even sub-range $\frac{r_t - l_t}{P}$. Such a partition function f is given in Eq. 1.

$$f(r_i) = \begin{cases} l_t \le r_i \le r_t & \lfloor \frac{(r_i - l_t) * P}{r_t - l_t} \rfloor \\ r_i \ge r_t & P - 1 \end{cases} \quad (1)$$

The functionalities of mappers and reducers are depicted in detail below.

- **Mapper**: Each mapper gets the time (resp. value) interval $[l_i, r_i]$ of one modeled segment i to check whether it intersects with the query time (resp. value) range. For the qualified modeled segments, the intermediate key is derived by the partition function $f(r_i)$. The model coefficients $< p_i^1, \cdots, p_i^n >$ are the value part of the intermediate key-value pair.
- **Reducer**: One reducer receives a list of qualified modeled segments $< p_0^1, \cdots, p_0^n >, < p_1^1, \cdots, p_1^n >, \cdots$. For each modeled segment $< p_i^1, \cdots, p_i^n >$, the reducer invokes a model-based gridding function to compute discrete values as query results.

Regarding the scan-based model filtering and gridding, as *SS*s in $\mathcal{S}_0$ are located in different regions of the index-model table, the query processor makes use of thread pool to process each *SS* of $\mathcal{S}_0$ in parallel. Fig. 4 shows the workflow of the hybrid KVI-Scan-MapReduce approach. For a time (or value) range query $[6, 10]$, $iSearch^+$ constructs the node set $\mathcal{S}_0 = \{15\}$ and $\mathcal{D} = [3, 11]$. Then, the *SS*s of the nodes in $\mathcal{D}$ are sent to MapReduce. The *SS* of $node15$, enclosed by the bottom dot-dashed block, is processed via range scan.

C. Theoretical analysis

One point to carefully consider is that $iSearch^+$ may generate redundant nodes, because the $iSearch^+$ aims to find a tight and consecutive range of *SS*s for MapReduce. For instance, in Fig. 4(a), the *SS* of $node4$ is not accessed by the $iSearch^+$, but is in the sub-table processed by MapReduce.

Theorem 1: For a range query $[l_t, r_t]$, the redundant nodes in $[l_s, r_s]$ returned by $iSearch^+$ are bounded.

Proof: Assume the height of the registration node τ as h. Consider the left-descending path from τ to the node l_0 closest to l_t. Let d be the depth from which the descending path turns right, namely the value $w < l_t$ of the current node. Then, based on the $iSearch^+$ algorithm, w is the left boundary of accessed node range and $w = \tau - \sum_{i=1}^{d} 2_{h-i}$.

For a certain value of d, the worst case happens when the descending process continues to go right until reaching l_0, as the nodes between w and l_0 are all redundant ones. The number of nodes returned by $iSearch^+$ under this case is given by:

$$U = \tau - (\tau - \sum_{i=1}^{d} 2^{h-i}) \quad (2)$$

The nodes between τ and w are all included into the output range $\mathcal{D}$ of $iSearch^+$. The number of nodes returned by the conventional *iSearch* is given by:

$$V = h - d + \{\tau - (\tau - \sum_{i=1}^{d} 2^{h-i} + \sum_{i=d+1}^{h} 2^{h-i})\} \quad (3)$$

Therefore, the number of redundant nodes returned by $iSearch^+$ is given by:

$$\begin{aligned} f = U - V &= d + \sum_{i=d+1}^{h} 2^{h-i} - h \\ &= d + 2^{h-d} - h - 1 \end{aligned} \quad (4)$$

The Eq. 4 is a function of d and is monotonous decreasing in d's domain $[1, h]$. Consequently, when $d = 1$, the function f reaches the maximum value, namely, the number of redundant nodes from $iSearch^+$ attains the maximum value. Therefore,

$$f_{max} = 2^{h-1} - h\,. \quad (5)$$

As the total number η of nodes of the sub-tree of the left child of τ is $2^h - 1$, hence

$$f \leq \frac{1}{2}\eta - log(\eta + 1) + \frac{1}{2}\,. \quad (6)$$

In summary, the total number of redundant nodes in the range $[l_s, r_s]$ is bounded. ■

The worst case happens when the endpoints l_t and r_t are the preceeding and succeeding nodes of τ, namely $l_t = \tau - 1$ and $r_t = \tau + 1$. However, for most of the cases, the redundant nodes returned from $iSearch^+$ are very limited.

VI. Experimental Evaluation

First, we compare model-view sensor data query processing with conventional one over raw sensor data. Then, we show that our KVI-Scan-MapReduce (*KSM*) approach outperforms other model-view sensor data querying approaches. Finally, we experimentally explore the factors that affect the performance of KVI-Scan-MapReduce.

A. Setup

We employ accelerometer data from mobile phones as sensor data set. The size of raw sensor data is 22 GB including 200 millions data points. After modeling, the modeled segments of the sensor data take 12 GB, while there are around 25 millions modeled segments.

We developed our system using the versions of Hbase and Hadoop in Cloudera CDH4.3.0. The experiments are performed on our own cluster that consists of 1 master node and 8 slaves. The master node has 64GB RAM, 3TB disk space (4 x 1TB disks in RAID5) and 12 cores, each of which is 2.30 GHz (Intel Xeon E5-2630). Each slave node has 6 cores 2.30 GHz (Intel Xeon E5-2630), 32GB RAM and 6TB disk space (3 x 2TB disks). Nodes are connected via 1GB Ethernet. In the experiment results, we refer to query selectivity as the ratio of the number of qualified modeled segments over that of total modeled segments.

Regarding the online sensor data model-based segmentation, we applied the *PCA*(piecewise constant approximation) [1]. Piecewise constant approximation (PCA) approximates a data segment with a constant value, which can be the mean value of the segment (referred to as the cache filter). We simulate sensor data emission and online time series segmentation [18] and upload them into the key-value store. Regarding gridding, we choose 1 second as the time granularity for discretization of a modeled segment. The gridding granularity is a parameter depending on end-user requirements.

B. Model-view sensor data vs. raw sensor data

Raw sensor data is a set of discrete data points each of which has associated timestamp and value. We create two tables, which respectively take the timestamp and sensor value as the row-keys, such that the query range or point can be used as keys to locate the qualified data points. Then, the query processor invokes the MapReduce to access the large size of data points for query results.

Fig. 5 (a), (b) and (c) present the query response times for time range, value range and point queries respectively. As shown in Fig. 5 (a) and (b), model-view approach takes around 30% less time than the raw sensor data method for both time and value range queries. Although the raw sensor data based methods apply MapReduce to directly access the qualified tuples via the row-key based range scan, the amount of raw sensor data to process is much larger than that of the model-view approach. In Fig. 5(c), the processing time of the raw data based method is $2\times$ less than that of the model-view one in time point queries, because the raw data method can use the query time point as index key to directly access the relevant data points, while our *KSM* requires to perform model filtering and gridding. For value point queries, the model-view approach has nearly $3\times$ less time than the raw data method. As normally there is a large size of data points with the query value, MapReduce is used to access this qualified sensor data set. In model-view approach, the point query processing only uses random access and range scan to get qualified modeled segments for gridding locally, and thus it saves the time on starting MapReduce to access data.

C. Comparison of model-view approaches

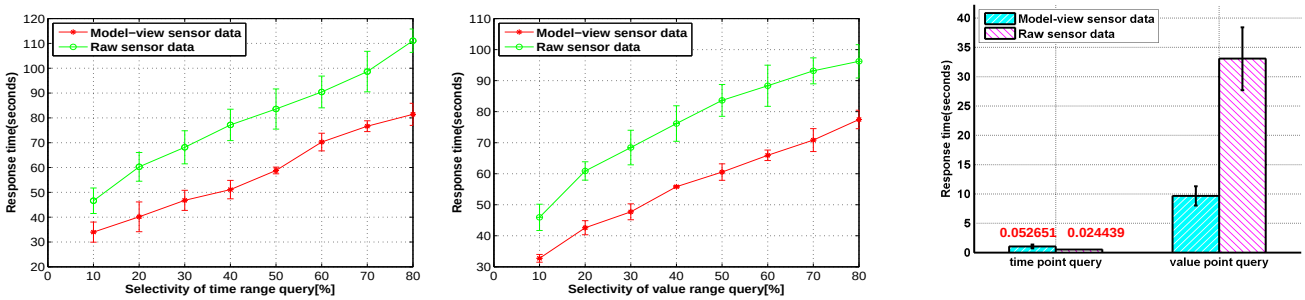
There are four baseline approaches for querying model-view sensor data, namely:

MapReduce (MR). This approach utilizes MapReduce without any support from index. It always works on the whole index-model table to filter the qualified modeled segments in the mapping phase and perform the model gridding in the reduce phase.

Interval tree (IT). We implemented the traditional query processing operations of the interval tree [6], [9] by adding another table to store the *SS* of each node sorted by the right end-point of intervals. Each index, time or value, has two associated tables. During the intersection or point search on *vs-tree*, the query processor decides which table to access based on the relation between the query range (or point) and the node value. In this way, the query processor can stop scanning once encountering one unqualified modeled segment, due to the monotonicity of end-points of modeled segments. *IT* makes use of random access and range scan to sequentially filter the qualified modeled segments and make gridding locally.

MapReduce+KVI (MRK). The idea of *MRK* is to leverage KVI-index to avoid having MapReduce to process the whole table. In *MRK*, MapReduce is designed to work over one continuous sub-index-model table including all the *SS*s of the accessed nodes in search operations. For instance, in Fig. 4(a), for a time (or value) range query $[6, 10]$, *MRK* invokes MapReduce to work on the sub-table within the row-key range $[< 3, \alpha >, < 16, \alpha >]$. The same idea applies for point queries. As compared to our hybrid *KSM* approach, *MRK* is a lightweight indexing-MapReduce plan, as it processes many irrelevant *SS*s of nodes between $\mathcal{S}_0$ and $\mathcal{D}$.

Filter of key-value store (FKV). Some key-value stores such as HBase provide a filter functionality to support predicate-based row selection. The filter transmits the filtering predicate to each region server and then all servers scan their local data in parallel. Afterwards, they return the qualified tuples. Our filter-based query processing also works on the index-model table, as the filtering predicates can be directly applied to the columns. The query processor waits until all



(a) Time range queries (b) Value range queries (c) Point queries

Fig. 5. Query performance of range and point queries

region servers finish scans and then it retrieves each returned qualified modeled segment to conduct gridding locally.

1) Range Query: Fig.6 (a), (b) and (c) present the performance of time range queries. As depicted in Fig.6(a), *KSM* outperforms *MR* up to $3\times$ for the low-selective time range queries. As the query selectivity increases, the amount of *SS*s for scan based processing decreases and that for MapReduce approaches the entire table. Therefore, the response time increases and approaches that of *MR*. The response time of *MR* increases little. As increasing query selectivity leads to ascending gridding workload in reduce phase, these results show that the overhead from model gridding is not dominant in *MR*. The response time of *MRK* is more than that of *KSM*, but less than that of *MR*. As *MRK* utilizes the KVI-index to localize a consecutive sub-index-model table covering all the *SS*s of nodes found by intersection search, it processes fewer modeled segments than *MR*'s full table scanning. Yet, as compared to *KSM*, *MRK* processes more redundant modeled segments. Moreover, as the sub-table in *MRK* covers a large range, the processing time of *MRK* increases little for low-selective queries.

Fig.6(b) exhibits the performance of *IT* and *FKV* approaches. As *FKV* needs to wait for each region server of HBase to finish the local data scanning, its total response time is a little longer than that of *IT* approach. They both consume much more time than all *MR*, *MRK* and *KSM*, as they apply sequential accessing of modeled segments.

We also analyse the number of modeled segments accessed by each approach in Fig. 6(c). These experiments show how different access methods of modeled segments affect the performance. *MR* works on the entire table, thus, the number of accessed segments is the same. From the point of view of the application, only qualified modeled segments are returned for gridding, thus *FKV* processes no redundant modeled segments and consumes the least amount of modeled segments. Since *IT* scans the *SS* of one node until encountering an unqualified model, the total number of accessed segments is a little larger than that of *FKV*. Our *KSM* processes larger number of segments than both *IT* and *FKV* due to the continuous and redundant range of *SS*s found by $iSearch^{+}$. However, the results also verify our theoretical analysis that the amount of redundant modeled segments is bounded. *MRK* accesses more segments than *IT*, *FKV* and *KSM*, as it adds the *SS*s between $\mathcal{S}_0$ and $\mathcal{D}$ to form a continuous sub-table for MapReduce. Referring to Fig. 6(a) and Fig. 6(b), although *KSM* approach consumes more segments than *IT* and *FKV*, its hybrid paradigm is the most efficient.

Fig. 6 (d), (e) and (f) present the value range query performance. The different query processing approaches exhibit similar patterns as for the time range queries, so we skip the detailed analysis.

2) Point Query: The time and value point query processing performance are shown in Fig. 6(g). *IT* wins both for time and value point queries. The response time of *KSM* is a little greater than *IT*, but outperforms the other approaches, because *IT* is able to access all qualified modeled segments in one *SS*. However, the *KSM* scans the whole *SS* entries of a node to find the qualified ones. Because of the invocation of MapReduce and redundant modeled segments in the sub-table, *MRK* takes more time than both *IT* and *KSM*. But, as *MRK* does not work on the entire table as *MR* does, it takes about $2\times$ less time than *MR*. *FKV* consumes the most time as it needs to wait for server-side full table scan before gridding operations. Since the size of the domain of the sensor data values is smaller than that of the time domain, nodes in value *vs-tree* accommodate more modeled segments than time *vs-tree* nodes. Thus, the response times of value point queries of *IT*, *FKV* and *KSM* approaches are all more than those of time point queries. For *MR* and *MRK*, the processing time differences between time and value queries are insignificant, as the time spent on model filtering and gridding is not dominant.

D. Insights into KVI-Scan-MapReduce

This experiment aims to reveal how much time *KSM* spends on model gridding, which is an additional step, as compared to querying raw sensor data. Due to space constraints, we only show the results from time range queries of selectivity from 10% to 50%. From Fig. 6(h), the time on model gridding accounts for $1/3$ - $1/2$ of the total processing time. As the majority of the gridding work is performed in the reduce phase and the amount of qualified segments for reducers depends on the query selectivity, the time spent on model gridding increases with the query selectivity. If the model gridding can adapt to users' different requirements for query results, the performance of *KSM* can be further improved.

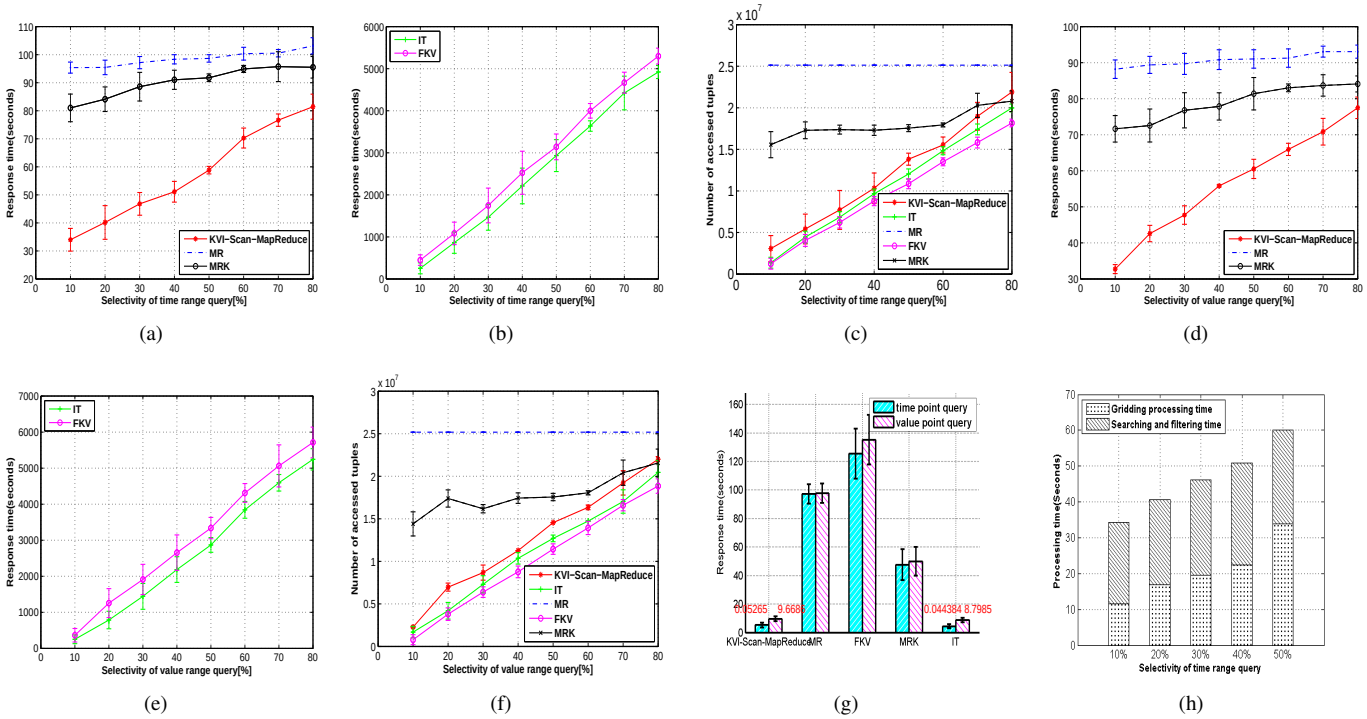


Fig. 6. Query performance (a)-(c): time range queries, (d)-(g): value range queries, (g) point queries, (h)query processing time constitution

VII. Conclusion

To the best of our knowledge, this is the first work to explore the key-value representation of an interval index for model-view based sensor data management. Different from conventional external-memory index structure with complex node merging and split mechanisms, our *KVI-index*, resident partially in memory and partially materialized in the key-value store, is easy to maintain in the dynamic sensor data generation environment. Moreover, we proposed a hybrid query processing approach, namely *KVI-Scan-MapReduce*, integrating the KVI-index, range scan and MapReduce for model-view sensor data in key-value stores. Extensive experiments in a real testbed showed that our approach outperforms in terms of query response time and index updating efficiency not only query processing methods based on raw sensor data, but also all other approaches considered based on model-view sensor data for time/value range and point queries. As a future work, we plan to explore how to process time and value composite queries based on KVI-index.

References

[1] S. Sathe, T. G. Papaioannou, H. Jeung, and K. Aberer, "A survey of model-based sensor data acquisition and management," in *Managing and Mining Sensor Data*. Springer, 2013.

[2] A. Thiagarajan and S. Madden, "Querying continuous functions in a database system," in *SIGMOD*, 2008.

[3] A. Deshpande and S. Madden, "Mauvedb: supporting model-based user views in database systems," in *SIGMOD*, 2006.

[4] T. Papaioannou, M. Riahi, and K. Aberer, "Towards online multi-model approximation of time series," in *MDM*, 2011.

[5] A. Bhattacharya, A. Meka, and A. Singh, "Mist: Distributed indexing and querying in sensor networks using statistical models," in *VLDB*, 2007.

[6] H.-P. Kriegel, M. Pötke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB*, 2000.

[7] "Opentsdb," in *http://opentsdb.net/*, 2011.

[8] C.-H. Ang and K.-P. Tan, "The interval b-tree," *Inf. Process. Lett.*, vol. 53, 1995.

[9] L. Arge and J. Vitter, "Optimal dynamic interval management in external memory," in *37th Annual Symposium on Foundations of Computer Science*, 1996.

[10] G. Sfakianakis, I. Patlakas, N. Ntarmos, and P. Triantafillou, "Interval indexing and querying on key-value cloud stores," in *ICDE*, 2013.

[11] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim, "The time index: An access structure for temporal data," in *VLDB*, 1990.

[12] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *MDM*, 2011.

[13] C. P. Kolovson and M. Stonebraker, "Segment indexes: dynamic indexing techniques for multi-dimensional interval data," *SIGMOD Rec.*, vol. 20, 1991.

[14] M.-Y. Iu and W. Zwaenepoel, "Hadooptosql: a mapreduce query optimizer," in *EuroSys*, 2010.

[15] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *VLDB Endow.*, vol. 3, 2010.

[16] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *VLDB Endow.*, vol. 5, 2012.

[17] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak, "Eagle-eyed elephant: split-oriented indexing in hadoop," in *EDBT*, 2013.

[18] T. Guo, Z. Yan, and K. Aberer, "An adaptive approach for online segmentation of multi-dimensional mobile data," in *Proc. of MobiDE, SIGMOD Workshop*, 2012.

[19] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *VLDB Endowment*, vol. 1, 2008.

[20] L. George, *HBase: the definitive guide*. O'Reilly Media, Incorporated, 2011.

Acknowledgment

This work is supported by the EU Project FP7-ICT-2011-7-287305 OpenIoT.